PreServe: Intelligent Management for LMaaS Systems via Hierarchical Prediction

Zhihan Jiang
The Chinese University of Hong Kong
Hong Kong SAR, China

Yujie Huang
The Chinese University of Hong Kong
Hong Kong SAR, China

Guangba Yu
The Chinese University of Hong Kong
Hong Kong SAR, China

Junjie Huang
The Chinese University of Hong Kong
Hong Kong SAR, China

Jiazhen Gu
The Chinese University of Hong Kong
Hong Kong SAR, China

Michael R. Lyu
The Chinese University of Hong Kong
Hong Kong SAR, China

ABSTRACT

Large Language Models (LLMs) have revolutionized numerous domains, driving the rise of Language-Model-as-a-Service (LMaaS) platforms that process millions of queries daily. These platforms must minimize latency and meet Service Level Objectives (SLOs) while optimizing resource usage. However, conventional cloud service management techniques, designed for traditional workloads, are suboptimal for LMaaS due to its dynamic service workloads and variable request loads. To address this, we propose PreServe, a tailored LMaaS management framework centered on hierarchical prediction. PreServe incorporates a *service workload predictor* to estimate periodic token density at a coarse granularity and a novel *request load predictor* to assess the resource demand of individual LLM requests, enabling the construction of a *load anticipator* for each LLM instance. By integrating both long-term and short-term predictions, PreServe adjusts resource allocation in advance, mitigating the risks of instance under- or over-provisioning. Besides, PreServe optimizes request routing by considering both current and anticipated future instance loads, ensuring balanced load distribution across instances. Evaluations on real-world production datasets show that PreServe outperforms state-of-the-art methods, reducing tail latency by 41.3%, cutting resource consumption by 49.38%, while incurring only 0.23% additional overhead.

1 INTRODUCTION

Large language models (LLMs) have emerged as transformative technologies in various domains, such as natural language processing [33, 80] and software engineering [28, 48]. Their applications in areas such as conversational systems [8, 14], search engines [22, 78], and code generation [10, 52] have influenced both daily life and professional workflows. These models are typically deployed under the **Language-Model-as-a-Service (LMaaS)** paradigm [43, 68], where users send requests (*i.e.*, prompts) to a service endpoint and receive the generated responses.

The widespread adoption of LLMs and their integration into diverse software [60, 75, 85] have led to LMaaS platforms processing millions of queries per day [54]. Similar to traditional software services, the effective management of these platforms is critical for delivering a high-quality user experience, ensuring service availability [71], and meeting Service Level Objectives (SLOs) [27].. LMaaS management pursues two interdependent goals: (1) provisioning sufficient resources to match demand without over- or under-provisioning and (2) routing LLM requests to appropriate instances to maintain load balance and minimize latency. Failure to achieve either goal degrades service performance, potentially violating SLOs and resulting in economic losses or user attrition [13].

Although advancements in traditional service management via auto-scaling [39, 47, 49, 56, 81] and request load balancing [17, 19, 58, 90] have enhanced management efficiency, the unique features of LMaaS reveal the limitations of these conventional approaches:

High workload variability exacerbates cold start issues. As characterized in § 3.1.1, real-world LLM services exhibit substantial fluctuations in tokens per second (TPS). While weekday patterns emerge, daily peak TPS values vary unpredictably, rendering direct service workload predictions inherently imprecise. Traditional auto-scaling compensates for such forecast errors by reactively adjusting instance counts based on real-time metrics (*e.g.*, CPU utilization exceeding 80%). While effective for conventional services with millisecond startup times, this reactive approach falters in LMaaS scenarios. The sheer scale of modern LLMs amplifies this issue. For instance, the Deepseek-R1 model [30] with 671 billion parameters incurs cold start times ranging from tens of seconds. Such delays make reactive scaling impractical, as new instances can hardly be provisioned swiftly enough to address sudden demand spikes.

High request load variance intensifies load imbalance. Unlike traditional cloud services, where requests of a given type exhibit consistent execution profiles, LLM request loads vary widely due to differences in response lengths. Our analysis of real-world LLM requests (§ 3.1.2) reveals significant heterogeneity: noting that resource consumption, particularly GPU memory, scales with the response length, unlike the more stable resource usage of traditional requests. This dynamic and imbalanced execution load undermines conventional load-balancing strategies [58, 90], such as round-robin and least-request approaches. Designed for uniform request profiles, these methods cannot adapt to the dynamic demands of LLM requests, leading to instance overloads.

These challenges underscore the necessity for management strategies tailored to the distinct dynamics of LLM services. To meet this need, we introduce PreServe, a hierarchical prediction-based LMaaS management framework aimed at optimizing resource utilization and reducing serving latency. PreServe consists of two main components: *hierarchical prediction* and *prediction-based management*. The *hierarchical prediction* component operates at two granularities. First, a *service workload predictor* (§ 4.1) leverages long-term patterns to forecast aggregate token demand. Second, a *request load predictor* (§ 4.2) estimates the individual request load in real time, informing a per-instance *load anticipator* (§ 4.3.1) of imminent load changes. The *prediction-based management* component uses these

forecasts to optimize operations: An *instance scaler* (§4.3.2) proactively adjusts resources to mitigate cold-start issues, while a *request router* (§4.3.3) uses anticipated loads to balance instance loads, reducing tail latency. This synergistic approach combines long-term workload trends with real-time request load dynamics, reducing dependency on reactive scaling, mitigating cold-start delays, and sustaining load equilibrium across LLM instances.

We evaluate PreServe using open-source traces from Microsoft Azure LLM services [65] and ShareGPT datasets [1]. Our results demonstrate that, compared to state-of-the-art baselines, PreServe reduces peak latency by over 45.1% while improving resource utilization by 49.38% in LMaaS systems under fluctuating workloads. Additionally, PreServe effectively routes LLM requests to maintain load balance across instances, leading to a reduction of over 41.3% in P99 normalized latency and 66.58% in SLO violations. Notably, PreServe introduces only a 0.23% overhead relative to request latency, underscoring its practical effectiveness.

In summary, this paper presents the following key contributions:

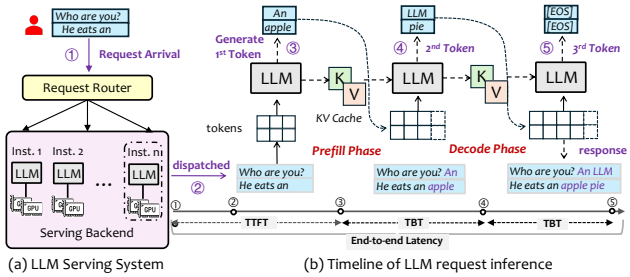
- Based on a detailed analysis of Azure LMaaS workloads, we identify two critical limitations of traditional service management techniques, creating opportunities for LMaaS solutions (§ 3).
- We introduce PreServe, a novel hierarchical LMaaS management framework to combine long-term workload periodicity with short-term request load dynamics, optimizing resource utilization and reducing serving latency (§ 4).
- Extensive evaluations indicate that PreServe outperforms all baselines, significantly reducing serving latency and resource consumption, with low extra overhead (§ 5). We also open-source PreServe at [11] to benefit both developers and researchers.

2 BACKGROUND

A typical LMaaS system, illustrated in the left part of Fig. 1, consists of a request router and a serving back-end with multiple LLM instances. The request router acts as the endpoint, receiving user requests and routing them to appropriate LLM instances within the serving back-end. Each LLM instance hosts a complete model replica, supported by a group of dedicated hardware resources (*e.g.*, GPUs), enabling optimized inference through serving engines such as vLLM [42], SGLang [87], and TensorRT-LLM [6]. These engines typically employ advanced batching strategies (*e.g.*, continuous batching [82]), which dynamically replace completed requests at each token step to maximize inference throughput.

Two-phase autoregressive generation. LLM services stream response tokens as they are generated. As shown in the right part of Fig. 1, LLM inference is autoregressive: the model iteratively generates one response token based on the preceding tokens. Specifically, the process begins with a single parallel computation over all input tokens in the prompt to produce the first token, known as the *prefill phase*, which is compute-intensive[89]. Subsequently, the inference enters the *decode phase*, where the model generates one new token per iteration using the prompt and previously generated tokens until reaching an End-of-Sequence (EOS) token or a predefined limit. Each generated token is immediately sent to the request router and streamed to the user in real time.

KV Cache and memory management. During inference, intermediate key and value tensors from the attention mechanism [70]



(a) LLM Serving System (b) Timeline of LLM request inference

Figure 1: a) The framework of LLM serving system and b) an example of LLM request inference procedure.

are reused to generate subsequent tokens. To avoid redundant computation, these tensors (KV cache [42]) are stored in GPU memory, making the *decode phase* highly memory-intensive [53, 55]. The KV cache size for each request grows with its token sequence length. For instance, processing 64 batched Llama3-8B requests [24] with 8K tokens each can require up to 64 GB of KV cache, compared to 15 GB for model parameters. Given that modern GPUs offer only tens of GBs of memory (*e.g.*, 80 GB on NVIDIA H100), memory capacity becomes the primary bottleneck for LLM serving. When GPU memory is exhausted, the inference engine must evict certain requests and free their KV cache; these requests are later reloaded, causing redundant recomputation and increased latency.

Metrics of LLM service quality. Traditional services [27, 34, 49] typically use end-to-end (E2E) latency as their Service Level Objective (SLO) metric However, due to the dynamic execution and streaming nature of LMaaS, this metric alone is insufficient. Recent work [53, 89] introduces alternative metrics better suited to LLM services: 1) *Time to First Token (TTFT)*: the duration from the request submission to the receipt of the first generated token. 2) *Time Between Tokens (TBT)*: the interval between sequentially generated tokens for each request. 3) *Normalized E2E latency*: the E2E latency divided by the total number of output tokens. These metrics collectively capture user experience: TTFT reflects responsiveness, TBT indicates streaming smoothness, and normalized E2E latency provides an overall efficiency measure.

3 MOTIVATION

3.1 Challenges in LMaaS Management

To investigate LMaaS management challenges, we analyze open-source invocation traces from two Azure production LLM services, *code* and *chat*, spanning one week and 44.1 million requests [65]. Each trace includes timestamps, prompt and response lengths. Unlike traditional request-based workloads, LLM workloads are inherently token-centric, as token throughput directly reflects computational demand. We further separate the analysis into prefill and decode phases due to their distinct resource characteristics.

3.1.1 Challenge 1: Temporal Variability in LLM Service Workloads. We first observe the workload variations across different LLM services from a service perspective. Fig. 2-(a) and (b) depict the average prompt and response tokens per second (TPS) for *code* and *chat*

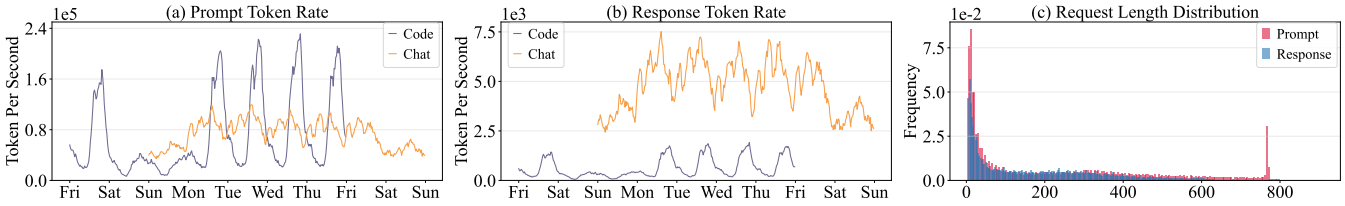


Figure 2: Azure LMaaS workload and ShareGPT datasets study: the distribution of the TPS for LLM (a) code and (b) chat service in Azure over one week, and (c) the prompt/response length in LLM conversations.

LLM services in Azure, aggregated over 30-minute intervals. Based on these trends, we identify three key observations:

(1) *LLM service workloads experience substantial fluctuations.* In the *code* service, the peak prompt TPS exceeds its average by 3.3× and its minimum by 35.6×, reflecting dramatic shifts within short timeframes. Such variability requires dynamic resource scaling to avoid performance degradation and resource waste.

(2) *Distinct LLM services exhibit different periodic workload patterns.* Both services demonstrate a clear diurnal trend: peaking during working hours and dropping at night and on weekends, enabling proactive forecasting. However, notable differences exist between services; for example, the peak prompt TPS for the *code* service are approximately twice those of the *chat* service, while the *chat* service's peak response TPS are nearly four times higher than those of the *code* service. These differences stem from their distinct application domains, indicating that LMaaS management must be tailored to each service.

(3) *Peak workload magnitudes introduce uncertainty.* Although weekday trends are consistent, daily peak TPS vary unpredictably (red lines in Fig. 2(a)–(b)) For instance, the highest prompt TPS peak of *code* service exceeds the lowest weekday peak by 35%. This uncertainty hampers the accuracy of workload predictions, complicating effective resource management.

Management Challenge 1: Service-specific provisioning heightens management intricacy, while extreme fluctuations and uncertain peak magnitudes strain the scalability and accuracy of existing management systems, thereby posing a risk of over-provisioning or degraded performance.

3.1.2 Challenge 2: Huge Variance of LLM Request Loads. In traditional cloud services, requests of a given type (*e.g.*, adding items or processing payments) exhibit similar execution profiles with minimal variability. Conversely, LLM service requests differ significantly due to variations in prompt complexity, necessitating a detailed analysis of inter-request differences. To quantify this variability, we analyzed token counts of prompts and responses using ShareGPT datasets [1], which consists of over 90,000 real-world conversations collected from OpenAI's chat service. We present their [P5, P95] distributions to mitigate noise, as shown in Fig. 2-(c).

Our analysis reveals substantial variability in LLM request token counts. Specifically, the prompt lengths vary significantly from 7 to 911 tokens, while responses range from 5 to 632 tokens, with median lengths of 52 and 87 tokens, respectively. This broad spectrum reflects the diverse nature of user interactions, ranging from brief queries to elaborate, multi-sentence dialogues. Such pronounced differences in prompt and response lengths translate directly into varying computational loads: longer prompts demand higher computation during the prefill phase, whereas extensive responses elevate memory usage and inference duration in the decode phase. Consequently, this variability undermines uniform request routing strategies, as heavy-load requests disproportionately consume resources, exacerbating load imbalances and potentially decreasing overall throughput, especially during peak usage periods.

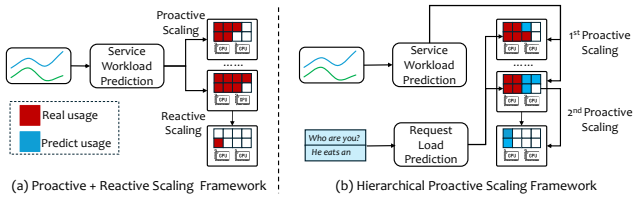


Figure 3: Different auto-scaling frameworks.

Management Challenge 2: The wide range of prompt and response lengths result in highly variable request loads. This complicates the design of effective request routing policies, as the uncertainty of individual request loads increases the risk of load imbalance, leading to increased serving latency.

3.2 Opportunities in LMaaS Management

The challenges of workload variability and diverse request loads highlight the limitations of traditional service management techniques and present opportunities for specialized LMaaS solutions.

3.2.1 Opportunity 1: Hierarchical Service Workload and Request Load Prediction. To cope with workload variability (Challenge 1), traditional services often use a hybrid scaling strategy (*e.g.*, Fig. 3-(a)): proactive scaling based on historical forecasts, supplemented by reactive scaling to handle prediction errors. This model, however, fails in the LMaaS context for two main reasons. First, the unpredictable workload peaks common in LMaaS lead to frequent forecast inaccuracies, necessitating reactive adjustments. However, the significant cold start latency of large LLM instances, often tens to hundreds of seconds [62], makes reactive scaling too slow to handle sudden demand spikes. Second, the high variance in per-request loads (Challenge 2) makes instance resource consumption highly volatile, triggering reactive scaling even more often and further exposing the impracticality of this approach.

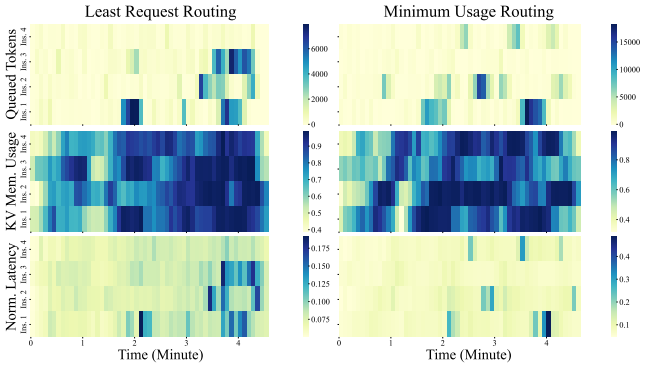


Figure 4: The performance timeline of two load-balancing algorithms on LMaaS benchmarks using four LLM instances.

These limitations highlight the need for a more proactive and adaptive strategy, naturally leading to a hierarchical approach combining service-level and request-level predictions (*i.e.*, Fig. 3-(b)):

(1) **Service-level Workload Prediction:** Historical TPS patterns (*e.g.*, workday peaks) forecast aggregate demand, enabling instance scaling in advance for upcoming time windows.
(2) **Request-level Load Prediction:** Real-time inference data estimate the remaining token loads per request, allowing early resource adjustments to prevent instance overload.

This hierarchical strategy addresses both long-term trends and short-term dynamics, reducing reliance on reactive scaling, mitigating cold start delays, and accommodating variable request loads.

Management Opportunity 1: Designing a hierarchical prediction framework that integrates service-level workload with request-level load prediction could improve resource efficiency amid temporal variability and request heterogeneity.

3.2.2 Opportunity 2: Predicted Load-aware Request Routing. The high variance in request loads (Challenge 2) invalidates traditional load-balancing policies [58, 90]. Strategies like round-robin or least-request, which assume uniform request loads, can hardly handle the dynamic and heterogeneous resource demands of LLM inference, causing severe load imbalance and degraded service quality [23, 51].

To illustrate these shortcomings, we conducted a pilot benchmark that evaluated two common routing strategies: least request (LR) and minimum use (MU) in an LMaaS environment. We deployed four LLaMA2-7B instances [12] on a 4× A40 GPU cluster and simulated user traffic using ShareGPT prompts [1] at a fixed queries-per-second (QPS) rate (9.5 in this case, a value corresponding to the overload boundary that has been widely adopted in prior research [57, 66, 76]). Three metrics were monitored: (1) *Queued Request Tokens*, the queued prompt tokens awaiting GPU computation; (2) *KV Memory Usage*, the percentage of GPU memory occupied by the KV cache, indicating decode-phase congestion; and (3) *Normalized E2E Latency*, a comprehensive quality metric defined in Sec 2. The LR strategy routed requests to the instance with the fewest active requests, while the MU strategy targeted the instance with the lowest weighted average of GPU utilization and memory usage. As shown in Fig. 4, both approaches have serious limitations.

(1) **LR:** At the 2-minute mark, instance 1 under LR exhibited a surge in prefill tokens, KV memory usage, and elevated normalized latency, signaling overload, while other instances remained underutilized. A similar imbalance recurred at 4 minutes. This stems from LR's reliance on request counts alone, ignoring the significant request load variability.
(2) **MU:** Despite factoring in utilization, MU still encountered instance overloads, such as instance 2 at 3 minutes and instance 1 at 4 minutes, due to its reactive nature. When an instance reaches saturation, the ongoing requests continue to generate responses, progressively elevating resource demands and resulting in persistent instance overload.

These findings underscore the critical need for routing strategies that explicitly account for LLM-specific load dynamics, moving beyond static or reactive paradigms. An effective policy should proactively anticipate current and future instance loads to mitigate potential load imbalance.

Management Opportunity 2: LMaaS providers can leverage the request-level load prediction to anticipate each instance's future request demands. This facilitates routing decisions that prevent instance overload, offering a proactive solution to the limitations of traditional load balancing.

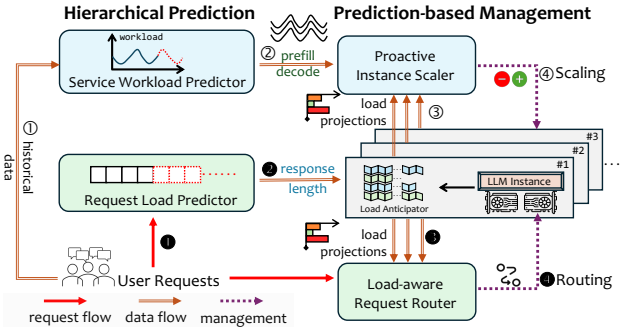


Figure 5: The overall framework of PreServe.

4 METHOD

Motivated by our insights, we propose PreServe, a hierarchical prediction-based LMaaS management framework, aiming to (1) dynamically autoscale LLM instances to optimize resource utilization under fluctuating workloads, and (2) predictively route LLM requests to achieve load balancing and reduce serving latency.

Fig. 5 overviews the workflow of PreServe, comprising two main parts: *hierarchical prediction* and *prediction-based management*. In long-term workload prediction, PreServe forecasts the periodic token density for future time windows and determines the appropriate number of LLM instances required (§ 4.1). For short-term load prediction, PreServe introduces a novel *request load predictor* designed to estimate dynamic loads of individual LLM requests (§ 4.2). For effective LMaaS management, the *instance scaler* integrates hierarchical prediction results to optimize resource allocation, preventing resource over- or under-provisioning (§4.3.2). Simultaneously, the *request router* considers both current and anticipated future loads across all instances when dispatching requests, ensuring instance load balancing and minimizing tail latency (§ 4.3.3).

Algorithm 1: Offline Phase of Service Workload Prediction

Input: Historical requests $\mathcal{R}$ and time windows $\mathcal{T} = \{T_1, T_2, \cdots\}$
Output: Prediction model $\mathcal{M}_P$, $\mathcal{M}_D$ and profiled rate μ_p, μ_d, μ_t

1 **for** $T_i \in \mathcal{T}$ **do**
2 Aggregate the prefill and decode tokens within $T_{i-k}, \cdots, T_i \Rightarrow$ Generate $\mathbf{P} = \{P_{i-k}, \ldots, P_i\}$, $\mathbf{D} = \{D_{i-k}, \ldots, D_i\}$
3 $\mathbf{P} = \frac{\mathbf{P} - \min(\mathbf{P})}{\max(\mathbf{P}) - \min(\mathbf{P})}$, $\mathbf{D} = \frac{\mathbf{D} - \min(\mathbf{D})}{\max(\mathbf{D}) - \min(\mathbf{D})}$ `// Normalization`
`/* Construct training datasets */`
4 Add training pair $\{P_{i-k}, \ldots, P_{i-1}\} \rightarrow P_i$ to S_P
5 Add training pair $\{D_{i-k}, \ldots, D_{i-1}\} \rightarrow D_i$ to S_D
`/* Update the maximum serving capability */`
6 **if** *all requests in* T_i *complete without an SLO violation* **then**
7 $\mu_p = \max\left(\mu_p, \frac{\sum p(r) \in T_i}{|T_i|}\right)$, $\mu_d = \max\left(\mu_d, \frac{\sum d(r) \in T_i}{|T_i|}\right)$
8 $\mu_t = \max\left(\mu_t, \frac{\sum (p(r)+d(r)) \in T_i}{|T_i|}\right)$
9 use S_P to train $\mathcal{M}_P$ and use S_D to train $\mathcal{M}_D$

Algorithm 2: Online Phase of Service Workload Prediction

Input: Historical aggregated token sequences: $\mathbf{P} = \{P_{i-k}, \cdots, P_{i-1}\}$, $\mathbf{D} = \{D_{i-k}, \cdots, D_{i-1}\}$
Output: Estimated N_1 LLM instances for the next time window

1 **for** *each current time window* T_i **do**
2 Predict current window tokens: $\hat{P}_i = \mathcal{M}_P(\mathbf{P})$, $\hat{D}_i = \mathcal{M}_D(\mathbf{D})$
`/* Extend historical sequences */`
3 $\mathbf{P'} = \mathbf{P} + \{\hat{P}_i\}$, $\mathbf{D'} = \mathbf{D} + \{\hat{D}_i\}$
4 Predict new window tokens: $\hat{P_{i+1}} = \mathcal{M}_P(\mathbf{P'})$, $\hat{D_{i+1}} = \mathcal{M}_D(\mathbf{D'})$
`/* Determine the required number of instances */`
5 $N_{i+1} = \max\left(\frac{\hat{P_{i+1}}}{\mu_p}, \frac{\hat{D_{i+1}}}{\mu_d}, \frac{\hat{P_{i+1}} + \hat{D_{i+1}}}{\mu_t}\right)$
6 **if** T_i *has concluded* **then**
7 Update historical sequences: $\mathbf{P} \leftarrow \mathbf{P} + \{P_i\}$, $\mathbf{D} \leftarrow \mathbf{D} + \{D_i\}$

4.1 Tier-1: Service Workload Prediction

As illustrated in Sec. 3.1.1, LLM service workloads exhibit dynamism, along with clear temporal periodicity. In light of this, PreServe introduces a *request workload predictor* to forecast long-term workload patterns and estimate the necessary number of LLM instances for upcoming coarse-grained time windows. Prior research [47, 49, 56, 83] has demonstrated the effectiveness of proactive auto-scaling strategies in optimizing the deployment of traditional services, including microservices and cloud services.

Nevertheless, the unique features of LLM services render simple prediction of request arrival patterns insufficient. First, requests from different LLM services can exhibit varying prompt and response token distributions. For instance, requests associated with conversational LLM services typically feature fewer prompt tokens and more response tokens compared to those for coding-related services. Second, due to the two-phase generation process in LLM inference, both the prompt (*i.e.*, the compute-bound prefill stage) and the response (*i.e.*, the memory-bound decode stage) can become performance bottlenecks in the serving system [53, 55]. Consequently, we propose forecasting the arrival patterns of both prompt and response tokens for each individual LLM service. This approach enables a more precise estimation of the number of LLM instances needed to handle future workload.

To accurately model and forecast token-level workload patterns of individual LLM services, we use the multiplicative Long Short-Term Memory (mLSTM) model [41]. The mLSTM integrates the strengths of Long Short-Term Memory (LSTM) networks [35], known for capturing both short-term and long-term patterns in time series data [46, 67], and multiplicative Recurrent Neural Networks (mRNN) [77], which enhance traditional RNNs with multiplicative transition functions to model complex dependencies and interactions. By incorporating the intermediate state of the mRNN with the gating mechanisms in LSTM, mLSTM achieves efficient, accurate, and robust time series predictions from historical data.

In particular, *service workload predictor* forecasts the prompt and response token densities for incoming requests within a fixed-length time window T_i, which is set to 10 minutes. Moreover, to determine the required number of LLM instances to serve specific token density, it also profiles the serving capability of each LLM instance from historical data. The application unfolds in two phases:

Offline training phase. During this phase, historical request data is collected for each LLM service over a brief period. This data includes request timestamps, prompt tokens, and response tokens, which are used to train the mLSTM model. The model takes as input the token counts from the previous k time windows, denoted $\{T_{i-k}, T_{i-k+1}, \ldots, T_{i-1}\}$, and predicts the token count for the subsequent time window T_i. To facilitate this, as shown in Alg. 1, for each time window T_i, we aggregate the total number of prompt and response tokens within the previous k windows (line 2). These collected pairs, *i.e.*, $\{P_{i-k}, \ldots, P_{i-1}\} \rightarrow P_i$ and $\{D_{i-k}, \ldots, D_{i-1}\} \rightarrow D_i$, are used to train the mLSTM models by learning the temporal patterns (line 4, 5, 9). Additionally, the predictor also profiles the maximum throughput of prefill, decode, and total tokens of all historical time windows for each instance without any SLO violations, which represents the prefilling, decoding, and hybrid serving capability of one LLM instance of the specific LLM service.

Online prediction phase. When applied in the online phase, the model generates predictions for future time windows in a stepwise manner. As illustrated in Alg. 2, at the beginning of each time window T_i , the mLSTM model predicts the token counts P_{i+1} and D_{i+1} for the subsequent time window T_{i+1} . This is accomplished through a two-step prediction process, where the model first predicts P_i and D_i for the current time window (line 2), then uses those predictions to forecast P_{i+1} and D_{i+1} for next time window (line 3, 4). Based on the projections of both prompt and response tokens, the predictor calculates the required LLM instances in the next time window, N_{i+1} (line 5). This look-ahead strategy facilitates the pre-initialization of LLM instances based on the projections, given that this initiation process typically requires dozens of seconds—well within the 10-minute time window. Upon completion of the current time window T_i, the ground truth values of P_i and D_i are appended to the input vectors $\mathbf{P}$ and $\mathbf{D}$, respectively, for the next prediction cycle (line 7). Additionally, the mLSTM model undergoes periodic re-training at longer intervals (*e.g.*, daily) using the most recent historical data. This re-training ensures that the model remains adaptive to shifting request patterns and trends of LLM services.

4.2 Tier-2: Request Load Prediction

While the workload predictor estimates the required number of LLM instances, prediction errors remain inevitable due to peak uncertainty and request burstiness [20, 21, 74]. Moreover, variability in LLM request loads complicates effective load balancing across instances (§ 3.1.2). To mitigate these issues, PreServe introduces request-level load prediction, which estimates the load of each individual request. This fine-grained, dynamic awareness of each LLM request's load allows PreServe to model and project the short-term load on each LLM instance. Consequently, PreServe can dynamically adjust LLM instances based on these immediate short-term load trends, rather than relying on historical data. This approach also facilitates predictive, load-aware request routing.

In detail, the load of each LLM request is influenced by both the prompt length, which is known beforehand, and the response length, which remains unknown until the request is completed. Recent studies [38, 57, 88] have demonstrated that response lengths for specific LLM services exhibit a strong correlation with the semantic content of query prompts. For example, translation requests typically yield response lengths comparable to the original prompt length. Thus, predicting response length emerges as a viable strategy for estimating the load of individual LLM requests. Nevertheless, designing such a predictor presents three main practical challenges:

(1) *Limited training data.* Compared to the broad input space of user prompts, only a small subset of LLM request data is available for training the predictor.
(2) *Strict latency requirements.* Given the real-time nature and stringent latency demands of LLM services, the predictor must operate with minimal latency to ensure its applicability.
(3) *Imbalance response length distribution.* As Fig. 2-(c) shows, LLM response lengths typically exhibit a long-tail distribution. Prior research [64] has proven that such imbalanced data can introduce biases and complicate the development of a predictor that achieves consistently high accuracy across diverse requests.

To tackle these challenges, we propose an effective and practical request load predictor, which is designed to take the prompt content as input and leverage its semantics to accurately predict the corresponding response length, *i.e.*, the number of decoded tokens.

In response to the limited training data, we select the pre-trained language model (PLM) as our proxy model, which is trained on large-scale corpora and has demonstrated strong semantic understanding capabilities [50, 73]. We also employ prompt tuning [45] to adapt PLMs to the specific task of response length prediction. This technique introduces a small set of additional parameters (*i.e.*, learnable prompt) into the model and optimizes these additional parameters rather than updating the entire model. Compared to traditional tuning approaches [32, 86], prompt tuning can effectively mitigate the risk of over-fitting with limited training data, thereby enhancing the model's performance in downstream tasks.

Specifically, given the stringent requirements for low overhead and latency, we select DistilBERT [59], a distilled version of BERT [40], which achieves a 60% faster runtime while retaining over 95% of BERT's performance on previous benchmarks [72]. As illustrated in Fig. 6, the final hidden state of the first token (*i.e.*, [CLS]) from the DistilBERT output is extracted as it serves as the aggregate sequence representation for the subsequent regression task [40].

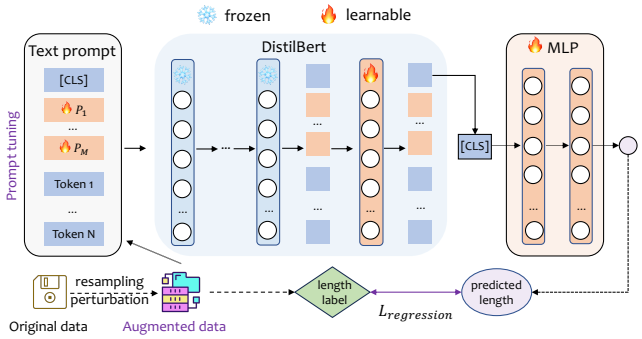


Figure 6: Request Load predictor training in PreServe.

This representation is passed through a two-layer feedforward neural network to generate the length prediction. During the prompt-tuning procedure, we append M additional learnable prompt tokens $P_1, P_2, \ldots, P_M$ to the input text tokens. These prompt tokens are learnable and adaptively updated during fine-tuning. To balance computational cost and performance, we also freeze the parameters of all DistilBERT layers except for the final layer.

To mitigate data imbalance issues, we adopt data augmentation techniques: *over sampling* [25] and *text perturbation* [79], to preprocess the original training data. First, we partition the training samples into N evenly spaced buckets based on response length, denoted as $\{B_1, B_2, \cdots, B_N\}$, where the bucket with the highest number of samples contains S samples. For buckets with fewer than μS samples ($\mu = 0.25$ in our experiments), we apply perturbation-based augmentation until the number of samples in each such bucket reaches μS. Specifically, for each sample selected for augmentation, a small proportion (*e.g.*, 15%) of words in the prompt are randomly replaced with their synonyms while maintaining the same response length label. Finally, the augmented dataset is used to train our request load predictor within a regression paradigm, consistently ensuring high accuracy for diverse LLM requests.

4.3 Prediction-based Management

Based on the hierarchical prediction results, PreServe effectively manages instance scaling and request routing within LLM services. Locally, each LLM instance maintains a *load anticipator* that forecasts its load over an upcoming period based on request load prediction. Centrally, the *instance scaler* leverages both long-term workload forecasts and short-term instance load projections to proactively scale instances. Concurrently, the *request router* accounts for both the immediate and anticipated loads of individual instances to optimize load balancing and maintain high service quality.

4.3.1 Instance Load Anticipator. Based on the predicted request loads, the load anticipator integrates a *load-look-ahead map* within each LLM instance, as illustrated on the left side of Fig. 7. This map estimates the KV cache memory usage for future iterations, motivated by the intuition that as the KV memory approaches saturation, preemption of requests will occur, which in turn affects the serving latency of LLM requests. Specifically, the load-look-ahead map records $\mathbf{U}$, the fraction of total KV tokens relative to total token capacity M, for the next L iterations, where L is the maximum

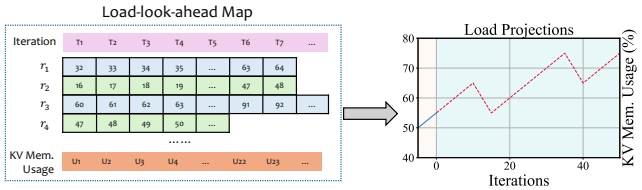


Figure 7: The load anticipator within each LLM instance.

output tokens of the LLM (*e.g.*, 4096 tokens for LLaMA-2). For a new LLM request r_n, with P prompt tokens and D predicted response tokens, the map is updated once the prefill phase is complete. This involves adjusting the map as follows: $U_i^{'} = \frac{U_i * M + (P+i)}{M}, i \in [0, D)$.

To mitigate inaccuracies in the predicted response length, where the ground truth response length $\hat{D}$ may differ from D, we refine the map to ensure more accurate projections. In detail, if a request completes earlier than predicted (*i.e.*, $\hat{D} < D$), the estimated token consumption for the remaining iterations is subtracted: $U_i^{'} = \frac{U_i * M - (P+i)}{M}, i \in [\hat{D}, D)$. Conversely, if a request does not complete within the predicted number of iterations (*i.e.*, $\hat{D} > D$), we introduce a "virtual" extension of the response length (set to $0.2\hat{D}$ in our implementation) and update the map accordingly. This process is repeated iteratively until the request completes, with any exceeded iterations being subtracted in the same manner.

Using the constructed load-look-ahead map, the load anticipator can project the instance's load over the next l iterations (*e.g.*, l = 100), as shown on the right side of Fig. 7, which serves as a guide for both instance scaling (§ 4.3.2) and request routing (§ 4.3.3).

4.3.2 Proactive Instance Scaler. As discussed in Sec. 3.2.1, to mitigate workload prediction inaccuracies and the prolonged cold-start delays associated with LLM services, PreServe proposes a hierarchical prediction-based mechanism for instance scaling. Specifically, this mechanism (1) periodically estimates the required number of instances for upcoming time windows based on long-term workload patterns and (2) continuously tracks short-term load projections of instances, dynamically adjusting the instance count accordingly. This two-tiered approach can effectively prevent both the over-provisioning and under-provisioning of LLM resources.

At the beginning of each prediction time window T_i, if the predicted instance count N_{i+1} exceeds the current count N_c, additional $N_{i+1} - N_c$ LLM instances are pre-initialized. Conversely, if $N_{i+1} < N_c$, the instance scaler conservatively scales down by isolating $N_c - N_{i+1}$ instances from the request routing process, allowing their ongoing requests to complete naturally.

During each time window, the instance scaler further adjusts the number of instances following a "one potentially overloaded instance, one additional instance scaled up" policy. An instance is regarded as potentially overloaded if, as predicted by its load anticipator for the next l iterations, its KV memory usage exceeds 95% in more than 10% of those iterations. This proactive adjustment mechanism enables earlier detection of overloads compared to purely reactive methods, thereby predictively scaling up to mitigate SLO violations. Moreover, to prevent excessive resource provisioning, a scaling-down action is activated if the predicted memory utilization across all instances remains consistently below a specified threshold, T_f (set at 30%), throughout the next l iterations. This scaling-down action will only be triggered once within each time window to avoid thrashing [84], with the number of isolated instances being $N_c - \frac{\sum_{ins.} \max(U^{'})}{T_f}$.

4.3.3 Load-aware Request Router. The *request router* in PreServe is designed to distribute LLM requests across multiple instances to mitigate instance overload and enhance serving efficiency by taking into account both current and anticipated future loads. Incoming requests are first subject to a queuing mechanism: if all instances are overloaded, the request is queued. If the queue capacity is surpassed, the request is aborted, and the user is notified. For each incoming request r with P prompt tokens and D response tokens to be predicted, the router estimates the potential load for each instance if request r is assigned to it. Specifically, the estimated load of i-th instance comprises three components: prefill load L_{p_i}, decode load L_{d_i}, and memory overflow risk L_{m_i} due to request evolving. The first two terms are computed as L_{p_i} = queued_$p_i + P$ and L_{d_i} = current_d_i + D, respectively, where queued_p_i represents total number of current queued token to be prefilled and current_d_i represent the total tokens to be generate in the decode phase. These two metrics reflect the immediate computational and memory demands on each instance.

To further mitigate the risk of KV memory overflow, the router queries each instance's load anticipator. The load anticipator "virtually" adds request r into its load-look-ahead map and returns the peak memory usage U_k anticipated over the next l iterations. Based on this projected usage, the router calculates a memory penalty component for the i-th instance as: $L_{m_i} = \max(0,\ U_k - T_{mem}) * M$,, where T_{mem} is the ideal memory usage that would not incur preemption latency (set to 80% in our experiments).

Finally, the request router dispatches r to the instance with the minimum estimated load by solving: $\arg\min_{i \in [1,N]} L_i = L_{p_i} + L_{d_i} + \beta \times L_{m_i}$, where β is the penalty term coefficient, which is set to 1.

5 EVALUATION

In this section, we conduct experiments to evaluate PreServe by answering the following research questions (RQs):

- **RQ1:** How accurate is PreServe in the hierarchical prediction?
- **RQ2:** How effective is PreServe in LLM instance scaling?
- **RQ3:** How effective is PreServe in LLM request routing?
- **RQ4:** What extra overhead does PreServe incur in management?

5.1 Evaluation Setup

Implementation. PreServe is implemented in 3.5K lines of Python code and designed in a non-intrusive paradigm, allowing seamless adaption to different serving frameworks and LLMs. In our experiments, we utilize the popular open-source LLM family, LLaMA [12], as our back-end LLMs. For the LLM inference engine within each LLM instance, we employ vLLM [15] (version 0.6.6), one of the most commonly used LLM serving frameworks [42].

Testbed. We conduct our experiments on an Ubuntu 22.04.1 GPU server configured with a 96-core Intel Xeon Gold 6342 CPU running at 2.80 GHz, 512 GB of RAM, and eight NVIDIA A40 GPUs, each with 40 GB of memory, connected via PCIe.

Table 1: Mean and maximum absolute percentage error (APE) of workload prediction in Azure datasets (10min-window).

Methods	Mean APE				Max APE			
	Azure-code		Azure-chat		Azure-code		Azure-chat	
	prompt	response	prompt	response	prompt	response	prompt	response
ARIMA	59.17%	61.44%	15.94%	16.12%	91.00%	90.18%	74.03%	82.09%
ETS	54.63%	55.55%	15.93%	16.10%	86.71%	83.54%	73.95%	81.96%
Prophet	26.26%	28.49%	8.05%	8.28%	67.88%	62.27%	27.30%	25.12%
PreServe	**7.74%**	**8.45%**	**4.15%**	**4.30%**	**26.25%**	**30.30%**	**21.16%**	**19.88%**
PreServe average	**8.10%**		**4.23%**		**28.28%**		**20.52%**	
	6.17%				**24.40%**			

Workloads. To emulate real-world LLM service invocations, we use the Azure LLM inference trace 2024 [65], which records 44 million requests from two production services, *azure-code* and *azure-chat*, over one week. Each trace entry includes timestamps and token counts for both prompts and responses, allowing us to reproduce request arrivals and enforce exact token generation by setting *ignore_eos* to true and *max_tokens* to the desired response length, following prior benchmarks [42, 44, 57, 74]. Since these traces omit prompt content, we further use the ShareGPT dataset [1], which provides over 90,000 real-world LLM conversations with detailed prompts and responses, to evaluate PreServe 's request router that predicts response lengths from prompt semantics.

Metrics. Our evaluation employs key metrics as introduced in Sec. 2 for assessing LLM service quality. In detail, we present results for *Time-to-First-Token (TTFT)* and *Normalized Latency*, which respectively represent the first token latency and the average latency per token. Additionally, we quantify Service Level Objective (SLO) attainment, defined as the proportion of requests fulfilled within the predetermined SLO threshold. Following prior studies [53], we set the SLO for Norm. Latency. It is set at 3× the normalized latency measured for a single, isolated request execution under no system contention. This value is determined by averaging the execution times of five independent runs, with the final SLO value being 0.2s.

5.2 RQ1: Accuracy of Hierarchical Prediction

5.2.1 Accuracy of Service Workload Predictor. We first assess the accuracy of the workload predictor in PreServe using Azure traces from *Azure-code* and *Azure-chat* services. To conduct this evaluation, we split the original dataset into a training set and a test set in a 1:1 ratio based on chronological order, ensuring that the former is used for model training while the latter is used for accuracy evaluation. The time window size is set to 10 minutes, which is slightly longer than the common scaling-up time of LLM instances (e.g., several minutes). For the evaluation metric, we adopt absolute percentage error (APE), defined as $\frac{|\text{prediction}-\text{actual}|}{\text{actual}}$. We report both the mean and maximum APE values across all time windows.

Baselines. We select several widely-used time series forecasting algorithms: *autoregressive integrated moving average* (ARIMA) [18], *exponential smoothing* (ETS) [29], and Prophet [69], for comparison. All these models are trained only using CPUs.

Results. The accuracies are presented in Tab. 1, demonstrating that PreServe consistently outperforms all baseline methods. Specifically, PreServe achieves an average mean APE of 8.10% and 4.23% for the code and chat services, surpassing the second-best baseline, Prophet, by 70.4% and 48.2%, respectively. This confirms the effectiveness of PreServe in effectively capturing periodic workload patterns of LLM services. Nevertheless, we also observe that all methods exhibit high maximum APE values, *e.g.*, ARIMA reaches a maximum APE of 91% in predicting prompt tokens for code service. Despite PreServe's superior performance, it still presents an average maximum APE of 24.4% for individual time windows. Such discrepancies can potentially lead to significant resource over- or under-provisioning, emphasizing the necessity of implementing hierarchical proactive scaling strategies.

Table 2: The response length prediction accuracy (up to 4096).

Methods	MAE	Acc-25	Acc-50	Acc-100
μ-Serve	355.59	32.31%	49.35%	65.25%
PiA (Vicuna-13B)	283.86	39.27%	54.18%	68.56%
PiA (ChatGPT)	127.41	50.42%	61.25%	70.34%
PreServe	**78.25**	**56.77%**	**68.79%**	**77.95%**
Improvement	↑ (38.6%)	↑ (12.6%)	↑ (12.3%)	↑ (10.8%)

5.2.2 Accuracy of Request Load Predictor. We quantitatively evaluate the accuracy of the request load predictor in estimating response lengths using the ShareGPT dataset, which is randomly divided into training and testing sets in a 8:2 ratio. The evaluation employs the mean absolute error (MAE) metric to measure the average difference between the predicted and actual number of tokens. Additionally, we consider three accuracy thresholds: *Acc-25*, *Acc-50*, and *Acc-100*, where a prediction is deemed correct if the difference is within 25, 50, or 100 tokens, respectively.

Baselines. We compare the request load predictor in PreServe with two state-of-the-art (SOTA) baselines: *μ-Serve* and *PiA*. μ-Serve [57] reformulates response length prediction as a classification task by dividing response lengths into multiple buckets. It fine-tunes BERT and a multilayer perceptron to classify LLM requests into these predefined length categories. We adopt the original implementation of μ-serve and use the median value of each bucket as the predicted length for its respective class. PiA (Perception in Advance)[88] modifies the original prompt by incorporating an instruction to estimate the response length in advance. For comparisons, we utilize two base LLMs: Vicuna-13B-v1.5[7] and ChatGPT [14].

Results. Tab. 2 presents the accuracy results. Despite employing the smaller DistilBERT model, PreServe significantly outperforms the BERT-based μ-Serve across all metrics, with a 78.0% improvement in MAE and a 19.5% improvement in Acc-100. These advances demonstrate the effectiveness of the training procedure in PreServe. Furthermore, PreServe also surpasses PiA when used with both Vicuna-13B and ChatGPT, achieving increases of 38.6% and 12.3% in MAE and Acc-50, respectively. The inferior performance of PiA stems primarily from its lack of fine-tuning to adapt the model from general language capabilities to the downstream response-length prediction task. In addition, PiA requires intrusive modifications to prompts and LLMs during inference, introducing substantial overheads that restrict its practical applicability. In contrast, PreServe optimizes the fine-tuning of a smaller, distilled model to balance the overhead and accuracy of response-length prediction.

5.3 RQ2: Effectiveness for Instance Scaling

Settings. In this RQ, we evaluate the effectiveness of PreServe's hierarchical proactive scaling strategy in ensuring service quality and optimizing resource efficiency. For experimental workloads,

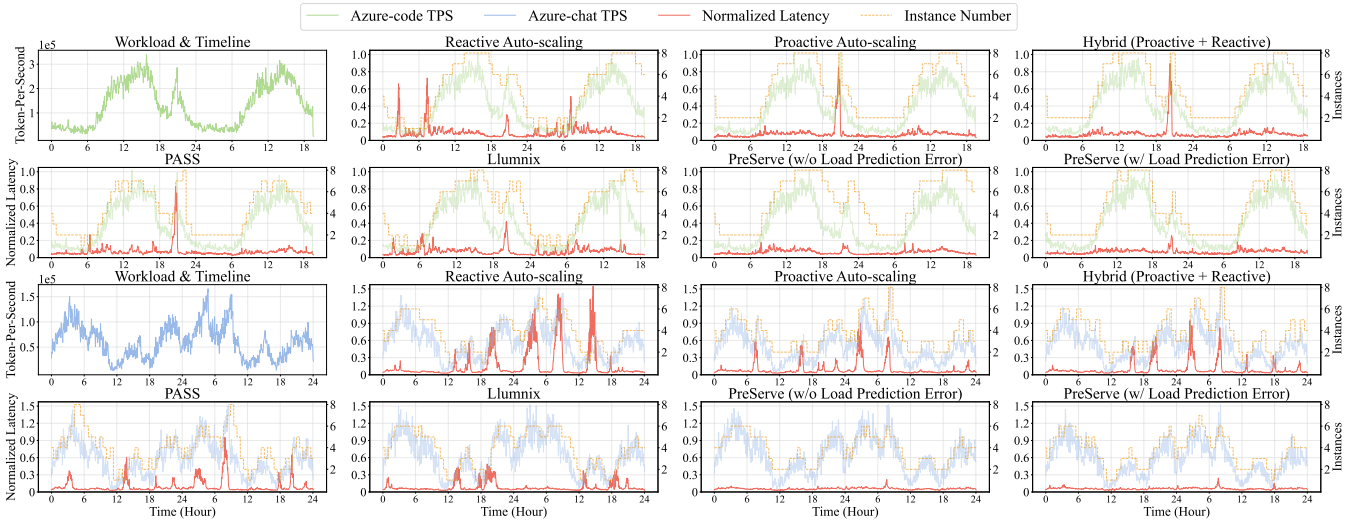


Figure 8: The evaluation results aggregated over 5-minute intervals under real-world workloads, using various scaling strategies.

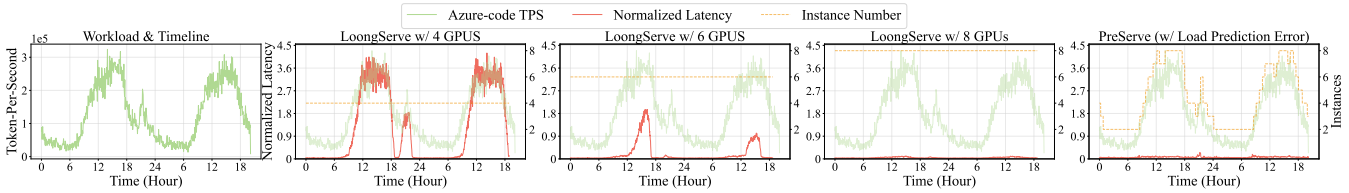


Figure 9: Evaluation of LoongServe compared to PreServe on the Azure-code workload across various resource configurations.

we utilize real-world traces from the *Azure-code* and *Azure-chat* services [65]. However, these publicly available traces typically omit prompt contents due to privacy concerns. To isolate the assessment of scaling capabilities, we first directly employ ground-truth response lengths for the request load predictor to independently assess scaling capabilities, a configuration we denote as PreServe w/o load errors. Besides, to simulate a more realistic scenario, we also introduce prediction errors based on the error proportion distribution from RQ1 Tab. 2, which we term PreServe w/ load errors. We use data from the first two days to train the workload predictor and the subsequent two days' data to generate the workload. To make the experiments tractable while preserving temporal patterns, we proportionally compress the two-day trace into approximately two hours. Lastly, we deploy up to eight LLaMA-2-7B instances and allow algorithms to dynamically auto-scale instances.

Baselines. We compare PreServe against five service-level management baselines: (1) *Reactive*: it leverages the current instance status (*i.e.*, KV memory usage) to reactively scale LLM instances up or down. (2) *Proactive*: it uses predicted LLM workloads to proactively determine the required number of LLM instances. (3) *Hybrid (proactive + reactive)*: it combines workload predictions and current instance status to dynamically determine the appropriate number of LLM instances. (4) *PASS* [31]: it combines workload forecasting with a serving performance model to proactively scale instances, aiming to minimize resource costs while meeting strict SLOs. For our comparison, we adopt its scaling strategy into the LMaaS instance scaler following the implementation described in the original paper. (5) *Llumnix* [66]: it introduces the concept of "virtual memory" for each LLM request and dynamically reschedules ongoing requests to prevent instance overload, as well as performs advance auto-scaling of instances. We use its open-source implementation [2] for our comparison. Furthermore, to distinguish our service-level approach from instance-level optimizations, we also compare PreServe against LoongServe [76], which dynamically adjusts the degree of parallelism for each request at runtime to improve serving efficiency. Since LoongServe is designed to operate on a fixed set of resources, we evaluate its performance on dynamic LMaaS workloads across different GPU allocations, using its open-source implementation [4].

Results of Service-level Management Comparison. We derive the following key observations based on Fig. 8:

- Reactive auto-scaling frequently introduces local latency spikes due to its untimely scaling response, primarily caused by the cold start time issue of LLM instances.
- Proactive, hybrid, and PASS, all fail to allocate instances in advance for unexpected workload spikes, *e.g.*, around 20 hours in the code service workload. In detail, PASS causes a high average peak normalized latency of 0.89s. This demonstrates that these approaches lack the capability to handle prediction errors.

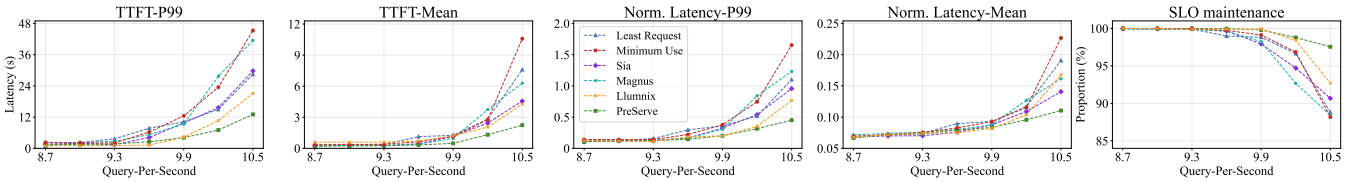


Figure 10: The latency and SLO maintenance of serving four LLM instances with different QPS under various routing strategies.

- While Llumnix is the best-performing baseline, its "virtual memory" mechanism and runtime request rescheduling still struggle with sudden workload surges. Consequently, it yields a high average peak latency of 0.455s. Furthermore, we observed frequent and unstable spikes in normalized request latency with Llumnix, which we attribute to the communication overhead and delays from KV cache migration during its runtime rescheduling.
- PreServe w/o load error achieves the best performance, maintaining a low average peak normalized latency of 0.197s. However, even when integrating the load error, PreServe still achieves superior performance, maintaining the average peak normalized latency within 0.250s. This robustness stems from two key factors: the high prediction accuracy of PreServe's load predictor and the error refinement mechanism within the load anticipator.

In summary, PreServe's hierarchical proactive strategy enables it to pre-allocate instances based on predicted workload patterns while adapting to unforeseen request spikes by anticipating instance loads. Compared to Llumnix, the top-performing baseline, PreServe reduces the peak normalized latency by an average of 45.1% to 0.250 seconds. Furthermore, when compared against a static eight-instance allocation, PreServe reduces resource consumption by an average of 49.38% with minimal SLO violations, still outperforming the SOTA Llumnix (which achieves a 48.26% reduction but suffers significant SLO violations). These results demonstrate PreServe's superior ability to enhance service quality and resource efficiency.

Results of Instance-level Inference Framework Optimization Comparison. Fig. 9 presents a performance comparison between PreServe, and LoongServe, a framework focused on instance-level optimization. With a static allocation of 4 GPUs, LoongServe's normalized latency reached 4.20s, significantly violating SLOs. Even with 6 GPUs, it could not accommodate workload peaks, leading to a peak latency of 1.96s. Although LoongServe's performance with 8 GPUs became comparable to that of PreServe, our system achieved this using only 4.35 GPUs on average, demonstrating superior resource efficiency. These results highlight that instance-level optimizations, which operate within a fixed resource paradigm, are inherently insufficient for the highly dynamic workloads of LMaaS. In contrast, PreServe focuses on service-level management to intelligently scale instances and route requests, which is essential for maintaining performance under fluctuating workloads. Furthermore, as PreServe's design is orthogonal to these instance-level approaches, they can also be integrated into the managed instances of PreServe for further gains.

5.4 RQ3: Effectiveness for Request Scheduling

Settings. In this research question (RQ), we evaluate the end-to-end performance of PreServe in routing requests to balance instance loads and reduce serving latency. Following previous studies [61, 66], we utilize prompt-response pairs from the ShareGPT dataset [1] to generate requests. These requests are produced at varying query-per-second (QPS) rates, following a Poisson distribution over a fixed duration (*e.g.*, 10 minutes).

Baselines. We select five widely-used and SOTA request routing strategies: (1) *least request (LR)* and *minimum use (MU)*: Two conventional load-balancing heuristics, and the detailed settings are provided in Sec. 3.2.2. (2) *Sia* [37]: A scheduler for ML clusters that optimizes job placement and resource allocation. We adapt it to learn a goodput model (mapping request features to instance throughput) from historical data and then use an integer linear programming (ILP) solver to determine the optimal routing policy, based on its implementation [5]. (3) *Magnus* [23]: A LMaaS scheduling framework that leverages predicted request output lengths. It groups requests with similar predicted lengths to minimize padding overhead. (4) *Llumnix* [3]: Following the RQ2 baseline setting, we evaluate LLumnix's request scheduling capability separately.

Results. The experimental results are presented in Fig. 10. We have made the following observations:

- When the QPS is low (*i.e.*, below 9.3), the system operates under a light load, and all routing strategies maintain comparable latency.
- As QPS increases, simple strategies like LR, MU, and Magnus fail to balance the dynamic LLM workloads, leading to high tail latency. For instance, at 10.5 QPS, Magnus reaches a P99 normalized latency of 1.23s, violating SLOs for 11.56% of requests. This is mainly because these methods are not designed for dynamic loads, and Magnus's routing logic is incompatible with modern continuous batching engines [82], where requests can dynamically join and leave batches.
- Sia's optimization-based routing policy, which lacks foresight into future loads, results in a P99 latency of 0.956s and only 90.68% SLO adherence, as the QPS reaches 10.5.
- Llumnix is the strongest baseline due to its ability to dynamically reschedule requests from overloaded to underloaded instances. However, this incurs substantial overhead from KV cache migration, leading to a considerable mean TTFT (4.26s), elevated P99 normalized latency (0.767s), and only 92.7% SLO adherence..
- In contrast, PreServe utilizes a predictive, load-aware routing strategy that considers both current system status and future instance load. This approach allows PreServe to maintain a low mean TTFT (2.24s), low P99 normalized latency (0.45s), and a minimal SLO violation rate (2.44%). These results correspond to performance improvements of 47.4%, 41.3%, and 66.58%, respectively, over the best-performing baseline, Llumnix.

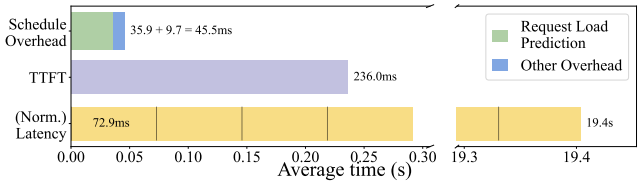


Figure 11: LMaaS management overhead of PreServe.

5.5 RQ4: LMaaS Management Efficiency

Settings. In this section, we evaluate the efficiency of PreServe in managing LMaaS services, focusing on its ability to maintain low overhead and high scalability. The primary cost associated with PreServe arises from the load prediction for each request and the maintenance of load anticipators for each LLM instance. To quantify this overhead, we measured the routing time for each LLM request under non-overloaded benchmark conditions, as specified in RQ3. The management overhead is defined as the interval between the arrival of a request at the router and its assignment to a specific LLM instance, encompassing both response length prediction and the subsequent routing decision. Moreover, we compared this overhead against metrics, including the average TTFT, normalized latency, and end-to-end latency across all served requests.

Results. Fig. 11 presents the aggregated average latency results for processed requests. Our analysis reveals that load prediction is the primary part of management overhead, averaging 35.9 ms per request. Other parts, such as load anticipator maintenance, collectively account for an additional 9.7 ms. In comparison, the average first token latency (TTFT) and per-token latency (normalized latency) stand at 236 ms and 72.9 ms, approximately 5.2× and 1.6× greater than the overhead, respectively. As depicted in Fig. 11, these latencies increase sharply to several thousand milliseconds and several hundred milliseconds when load imbalance occurs among instances. Furthermore, the average end-to-end latency across all processed requests is 19.4 seconds, exceeding the scheduling overhead by three orders of magnitude. Notably, PreServe introduces an average additional latency of merely 0.23% relative to the e2e request latency, yet it effectively mitigates latency spikes caused by load imbalance, underscoring its practical effectiveness.

6 THREATS TO VALIDITY

Evaluated models and serving frameworks. We implemented and evaluated PreServe with the LLaMA model family [12] and the vLLM framework [15], given their popularity and extensive use in recent studies [36, 44, 66]. However, PreServe is designed to be non-intrusive and can be readily integrated with various LLMs and inference frameworks. Furthermore, since PreServe is not tied to specific hardware architectures and operates mainly at the software service level, its broad applicability is assured. The core principles PreServe relies on, such as autoregressive generation and iteration-based inference, are fundamental across diverse LLM frameworks, ensuring its effective generalization to various LMaaS platforms.

Reliance on Historical Request Data. PreServe relies on historical LMaaS request data to train the hierarchical predictor and profile the serving capabilities of LLM instances. The absence of such data at the initial stage of service deployment presents a "cold start" challenge. This limitation is, however, transient; the predictors require a minimal volume of training data, often corresponding to a few days of service operation. Throughout this preliminary phase, default scaling and routing protocols can be implemented pending the accumulation of adequate data. Furthermore, to accommodate the dynamic characteristics of the LLM service, the predictors can be periodically retrained.

7 RELATED WORK

We structure our literature review in the following two domains:

(1) Service-level Management. Service management and SLO maintenance have been extensively studied in traditional cloud and web systems [21, 27, 31, 56], primarily focusing on two strategies: automated resource provisioning (*i.e.*, scaling) [16, 39, 62, 71] and dynamic request routing (*i.e.*, load balancing) [17].

(1.a) Instance Auto-scaling. In traditional cloud services, predictive auto-scaling is a well-established technique, using workload forecasts to proactively provision resources [31]. However, the significant instance startup times in LMaaS [26] render purely reactive scaling ineffective. To address this, some recent works focus on vertical scaling; for example, μ-Serve [57] dynamically adjusts GPU frequencies to handle minor load fluctuations. Other works employ reactive horizontal scaling. Llumnix [66], for instance, mitigates overload by migrating in-flight requests to new instances, though this incurs communication overhead. In contrast, PreServe advances the state-of-the-art by introducing a hierarchical prediction mechanism. This allows for fine-grained load projections, enabling proactive horizontal scaling that preempts congestion and avoids the overhead of reactive migration.

(1.b) Request Routing. Effective request routing is critical for performance. Early systems like DeepSpeed-MII [9] used simple round-robin policies. PiA [88] and Magnus [23], for example, improve performance by grouping requests with similar predicted output lengths. However, this approach is often incompatible with the widely-used continuous batching technique [82]. Other works like Sia [37] apply optimization techniques, using ILP solvers to find optimal job placements in general ML clusters. PreServe contributes a novel routing strategy that leverages its predictive hierarchy to intelligently dispatch requests, achieving superior load distribution while remaining compatible with modern serving frameworks.

(2) Instance-Level LLM Inference Optimization A distinct but complementary line of research focuses on optimizing the performance of a single LLM inference instance on fixed hardware. These works aim to boost throughput or reduce latency by optimizing the inference engine. For example, some methods disaggregate prefill and decode phases onto separate hardware (DistServe [89]), enable dynamic sequence parallelism (LoongServe [76]), or use hybrid CPU-GPU execution based on neuron activation locality (PowerInfer [63]). These works are orthogonal to PreServe, as it operates at a higher service management layer, orchestrating entire instances, whereas these methods optimize operations *within* an instance. As such, these approaches are complementary: the performance gains from an optimized inference engine can be further amplified by PreServe's intelligent service-level scaling and routing.

8 CONCLUSION

In this paper, we introduce PreServe, an LMaaS management framework that addresses key challenges in LMaaS management. PreServe employs hierarchical predictions, using a service workload predictor for periodic token density forecast and a request load predictor for per-request load estimation. By combining long- and short-term predictions, PreServe proactively adjusts resource provision and optimizes request routing to balance instance loads. Evaluations on real-world LMaaS datasets show that PreServe outperforms all baselines, significantly reducing latency and resource consumption with minimal overhead. We believe PreServe will contribute to advancing effective LLM service management.

ACKNOWLEDGMENTS

This work was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. SRFS2425-4S03 of the Senior Research Fellow Scheme and No. CUHK 14209124 of the General Research Fund).

REFERENCES

[1] 2023. ShareGPT Datasets. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered
[2] 2024. LLumnix Implementation. https://github.com/AlibabaPAI/llumnix
[3] 2024. LLumnix Implementation. https://github.com/AlibabaPAI/llumnix
[4] 2024. LoongServe Implementation. https://github.com/LoongServe/LoongServe
[5] 2024. Sia Implementation. https://github.com/siasosp23/artifacts
[6] 2024. TensorRT-LLM. https://nvidia.github.io/TensorRT-LLM/
[7] 2024. Vicuna-13B-v1.5. https://huggingface.co/lmsys/vicuna-13b-v1.5
[8] 2025. DeepSeek Chat. https://chat.deepseek.com/
[9] 2025. Deepspeed-mii. https://github.com/microsoft/DeepSpeed-MII
[10] 2025. Github Copilot. https://github.com/features/copilot
[11] 2025. The implementation repository of PreServe. https://github.com/OpsPAI/PreServe
[12] 2025. The Llama Family. https://huggingface.co/meta-llama
[13] 2025. OpenAI API and ChatGPT Performance Degradation. https://status.openai.com/incidents/01JMYB4A62XNXSAHAY8ZGQAAKD
[14] 2025. OpenAI ChatGPT. https://openai.com/chatgpt/overview/
[15] 2025. vLLM. https://github.com/vllm-project/vllm
[16] Hussain Ahmad, Christoph Treude, Markus Wagner, and Claudia Szabo. 2025. Towards resource-efficient reactive and proactive auto-scaling for microservice architectures. *Journal of Systems and Software* 225 (2025), 112390.
[17] Ratnadeep Bhattacharya, Yuan Gao, and Timothy Wood. 2024. Dynamically balancing load with overload control for microservices. *ACM Transactions on Autonomous and Adaptive Systems* 19, 4 (2024), 1–23.
[18] George EP Box and David A Pierce. 1970. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *Journal of the American statistical Association* 65, 332 (1970), 1509–1526.
[19] Hongchen Cao, Xinrui Liu, Hengquan Guo, Jingzhu He, and Xin Liu. 2024. QueueFlower: Orchestrating Microservice Workflows via Dynamic Queue Balancing. In *2024 IEEE International Conference on Web Services (ICWS)*. IEEE, 1293–1299.
[20] Giuliano Casale, Amir Kalbasi, Diwakar Krishnamurthy, and Jerry Rolia. 2011. Burn: Enabling workload burstiness in customized service benchmarks. *IEEE Transactions on Software Engineering* 38, 4 (2011), 778–793.
[21] Tao Chen and Rami Bahsoon. 2016. Self-adaptive and online qos modeling for cloud-based software services. *IEEE Transactions on Software Engineering* 43, 5 (2016), 453–475.
[22] Zehui Chen, Kuikun Liu, Qiuchen Wang, Jiangning Liu, Wenwei Zhang, Kai Chen, and Feng Zhao. 2024. MindSearch: Mimicking Human Minds Elicits Deep AI Searcher. arXiv:2407.20183 [cs.CL] https://arxiv.org/abs/2407.20183
[23] Ke Cheng, Wen Hu, Zhi Wang, Peng Du, Jianguo Li, and Sheng Zhang. 2024. Enabling efficient batch serving for lmaas via generation length prediction. In *2024 IEEE International Conference on Web Services (ICWS)*. IEEE, 853–864.
[24] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).
[25] Andrew Estabrooks, Taeho Jo, and Nathalie Japkowicz. 2004. A multiple resampling method for learning from imbalanced data sets. *Computational intelligence* 20, 1 (2004), 18–36.
[26] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM:Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 135–153.
[27] Alessio Gambi and Giovanni Toffetti. 2012. Modeling cloud performance with kriging. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1439–1440.
[28] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R Lyu. 2023. What makes good in-context demonstrations for code intelligence tasks with llms?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 761–773.
[29] Everette S Gardner Jr. 1985. Exponential smoothing: The state of the art. *Journal of forecasting* 4, 1 (1985), 1–28.
[30] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).
[31] Yunda Guo, Jiake Ge, Panfeng Guo, Yunpeng Chai, Tao Li, Mengnan Shi, Yang Tu, and Jian Ouyang. 2024. Pass: Predictive auto-scaling system for large-scale enterprise web applications. In *Proceedings of the ACM Web Conference 2024*. 2747–2758.
[32] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogerio Feris. 2019. Spottune: transfer learning through adaptive fine-tuning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 4805–4814.
[33] Zhankui He, Zhouhang Xie, Rahul Jha, Harald Steck, Dawen Liang, Yesu Feng, Bodhisattwa Prasad Majumder, Nathan Kallus, and Julian McAuley. 2023. Large language models as zero-shot conversational recommenders. In *Proceedings of the 32nd ACM international conference on information and knowledge management*. 720–730.
[34] Safiollah Heidari and Rajkumar Buyya. 2019. A cost-efficient auto-scaling algorithm for large-scale graph processing in cloud environments with heterogeneous resources. *IEEE Transactions on Software Engineering* 47, 8 (2019), 1729–1741.
[35] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[36] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, et al. 2024. Memserve: Context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565* (2024).
[37] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 642–657.
[38] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S3: Increasing GPU Utilization during Generative Inference for Higher Throughput. *Advances in Neural Information Processing Systems* 36 (2023), 18015–18027.
[39] Joao Paulo Karol Santos Nunes, Shiva Nejati, Mehrdad Sabetzadeh, and Elisa Yumi Nakagawa. 2024. Self-adaptive, requirements-driven autoscaling of microservices. In *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 168–174.
[40] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, Vol. 1. Minneapolis, Minnesota, 2.
[41] Ben Krause, Liang Lu, Iain Murray, and Steve Renals. 2016. Multiplicative LSTM for sequence modelling. *arXiv preprint arXiv:1609.07959* (2016).
[42] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.
[43] Emanuele La Malfa, Aleksandar Petrov, Simon Frieder, Christoph Weinhuber, Ryan Burnell, Raza Nazar, Anthony Cohn, Nigel Shadbolt, and Michael Wooldridge. 2024. Language-Models-as-a-Service: Overview of a New Paradigm and its Challenges. *Journal of Artificial Intelligence Research* 80 (2024), 1497–1523.
[44] Malgorzata Lazuka, Andreea Anghel, and Thomas Parnell. 2024. LLM-Pilot: Characterize and Optimize Performance of your LLM Inference Services. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–18.
[45] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691* (2021).
[46] Youru Li, Zhenfeng Zhu, Deqiang Kong, Hua Han, and Yao Zhao. 2019. EA-LSTM: Evolutionary attention-based LSTM for time series prediction. *Knowledge-Based Systems* 181 (2019), 104785.
[47] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2022. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*. 355–369.
[48] Michael R Lyu, Baishakhi Ray, Abhik Roychoudhury, Shin Hwei Tan, and Patanamon Thongtanunam. 2024. Automatic Programming: Large Language Models and Beyond. *arXiv preprint arXiv:2405.02213* (2024).

[49] Chunyang Meng, Shijie Song, Haogang Tong, Maolin Pan, and Yang Yu. 2023. DeepScaler: Holistic Autoscaling for Microservices Based on Spatiotemporal GNN with Adaptive Graph Learning. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 53–65.

[50] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. 2023. Recent advances in natural language processing via large pre-trained language models: A survey. *Comput. Surveys* 56, 2 (2023), 1–40.

[51] Chengyi Nie, Rodrigo Fonseca, and Zhenhua Liu. 2024. Aladdin: Joint Placement and Scaling for SLO-Aware LLM Serving. *arXiv preprint arXiv:2405.06856* (2024).

[52] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An Empirical Comparison of Pre-Trained Models of Source Code. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, 2136–2148. https://doi.org/10.1109/ICSE48619.2023.00180

[53] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

[54] Yifan Qiao, Shu Anzai, Shan Yu, Haoran Ma, Yang Wang, Miryung Kim, and Harry Xu. 2024. ConServe: Harvesting GPUs for Low-Latency and High-Throughput Large Language Model Serving. arXiv:2410.01228 [cs.DC] https://arxiv.org/abs/2410.01228

[55] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation—A KVCache-centric Architecture for Serving LLMChatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 155–170.

[56] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 805–825.

[57] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. 2024. Power-aware Deep Learning Model Serving with mu-Serve. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 75–93.

[58] Rasmus V Rasmussen and Michael A Trick. 2008. Round robin scheduling–a survey. *European Journal of Operational Research* 188, 3 (2008), 617–636.

[59] V Sanh. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[60] Yuchen Shao, Yuheng Huang, Jiawei Shen, Lei Ma, Ting Su, and Chengcheng Wan. 2025. Are LLMs Correctly Integrated into Software Systems?. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 741–741.

[61] Jiacheng Shen, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. 2021. Defuse: A dependency-guided function scheduler to mitigate cold starts on faas platforms. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 194–204.

[62] Shijie Song, Haogang Tong, Chunyang Meng, Maolin Pan, and Yang Yu. 2024. FuncScaler: Cold-Start-Aware Holistic Autoscaling for Serverless Resource Management. In *2024 IEEE International Conference on Web Services (ICWS)*. IEEE, 1036–1047.

[63] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 590–606.

[64] Samuel Stocksieker, Denys Pommeret, and Arthur Charpentier. 2023. Data augmentation for imbalanced regression. *arXiv preprint arXiv:2302.09288* (2023).

[65] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. Dynamollm: Designing llm inference clusters for performance and energy efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1348–1362.

[66] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 173–191.

[67] Luming Sun, Shijin Gong, Tieying Zhang, Fuxin Jiang, Zhibing Zhao, Jianjun Chen, and Xinyu Zhang. 2023. SUFS: A generic storage usage forecasting service through adaptive ensemble learning. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3168–3181.

[68] Tianxiang Sun, Yunfan Shao, Hong Qian, Xuanjing Huang, and Xipeng Qiu. 2022. Black-box tuning for language-model-as-a-service. In *International Conference on Machine Learning*. PMLR, 20841–20855.

[69] Sean J Taylor and Benjamin Letham. 2018. Forecasting at scale. *The American Statistician* 72, 1 (2018), 37–45.

[70] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[71] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. 2021. A Kubernetes controller for managing the availability of elastic microservice based stateful applications. *Journal of Systems and Software* 175 (2021), 110924.

[72] Alex Wang. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* (2018).

[73] Haifeng Wang, Jiwei Li, Hua Wu, Eduard Hovy, and Yu Sun. 2023. Pre-trained language models and their applications. *Engineering* 25 (2023), 51–65.

[74] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, et al. 2024. BurstGPT: A Real-world Workload Dataset to Optimize LLM Serving Systems. *arXiv preprint arXiv:2401.17644* (2024).

[75] Irene Weber. 2024. Large language models as software components: A taxonomy for llm-integrated applications. *arXiv preprint arXiv:2406.10300* (2024).

[76] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 640–654.

[77] Yuhuai Wu, Saizheng Zhang, Ying Zhang, Yoshua Bengio, and Russ R Salakhutdinov. 2016. On multiplicative integration with recurrent neural networks. *Advances in neural information processing systems* 29 (2016).

[78] Haoyi Xiong, Jiang Bian, Yuchen Li, Xuhong Li, Mengnan Du, Shuaiqiang Wang, Dawei Yin, and Sumi Helal. 2024. When Search Engine Services Meet Large Language Models: Visions and Challenges. *IEEE Transactions on Services Computing* 17, 6 (2024), 4558–4577. https://doi.org/10.1109/TSC.2024.3451185

[79] Zekun Xu, Abhinav Aggarwal, Oluwaseyi Feyisetan, and Nathanael Teissier. 2020. A differentially private text perturbation method using a regularized mahalanobis metric. *arXiv preprint arXiv:2010.11947* (2020).

[80] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. 2024. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *ACM Transactions on Knowledge Discovery from Data* 18, 6 (2024), 1–32.

[81] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2020. Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach. *IEEE Transactions on Cloud Computing* 10, 2 (2020), 1100–1116.

[82] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[83] Matineh ZargarAzad and Mehrdad Ashtiani. 2023. An auto-scaling approach for microservices in cloud computing environments. *Journal of Grid Computing* 21, 4 (2023), 73.

[84] Zhiyu Zhang, Tao Wang, An Li, and Wenbo Zhang. 2022. Adaptive auto-scaling of delay-sensitive serverless services with reinforcement learning. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 866–871.

[85] Yanjie Zhao, Xinyi Hou, Shenao Wang, and Haoyu Wang. 2025. Llm app store analysis: A vision and roadmap. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–25.

[86] Hongling Zheng, Li Shen, Anke Tang, Yong Luo, Han Hu, Bo Du, and Dacheng Tao. 2023. Learn from model beyond fine-tuning: A survey. *arXiv preprint arXiv:2310.08184* (2023).

[87] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

[88] Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. 2024. Response length perception and sequence scheduling: An llm-empowered llm inference pipeline. *Advances in Neural Information Processing Systems* 36 (2024).

[89] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210.

[90] Liangshuai Zhu, Jianming Cui, and Gaofeng Xiong. 2018. Improved dynamic load balancing algorithm based on Least-Connection Scheduling. In *2018 IEEE 4th Information Technology and Mechatronics Engineering Conference (ITOEC)*. IEEE, 1858–1862.